Large-scale scientific application programs in chemistry and physics on an experimental parallel computer system

by G. Corongiu
J. H. Detrich

We present and discuss an experimental distributed system consisting of two IBM 4341s, an IBM 4381, and ten FPS-164 attached processors, configured to allow parallel execution of a single large-scale calculation on multiple processors. A number of our application programs have been converted to run on this system, and the strategy for this conversion is outlined in sufficient detail to facilitate the development of tests using other scientific and engineering computer application programs. Our tests, though limited to certain biochemical and physicochemical problems, demonstrate the versatility, flexibility, and accessibility of this system, and we find performance comparable to that of today's supercomputers, which suggests that our approach can provide a practical answer to many large-scale computer applications.

1. Introduction

Scientific and engineering computer applications are rich and varied, but share some common needs. Success in these applications invariably gives impetus to the development of more elaborate calculations, so there is constant pressure to expand to the limits of currently available computer resources and beyond. In response to this demand, somewhat specialized hardware and software have been developed to extend the range of feasible calculations. One example is the advent of array processors, which emerged to handle signal and image processing but have found use in many other applications. Another example is the development of very fast, powerful, vector-oriented processors such as the CRAY 1S or X-MP and the CDC CYBER 205. One result of the successful use of these resources has been to bring into view scientific and

engineering calculations that are beyond the capability of even these supercomputers.

Our laboratory has always participated in this process, since our research interests center on problems in theoretical chemistry and biophysics that can only be resolved with the help of large-scale computer calculations. Hence computer hardware and software are crucial resources for our research, and we naturally strive to upgrade and expand them as much as possible. This is a primary motivation for the experiments in parallel systems that we report in this paper. Our goal is a computer system that is 1) at least as fast and possibly faster than supercomputers such as the CRAY 1S or CDC CYBER 205, 2) more flexible and versatile, 3) extendable to very high capability, and 4) not too expensive.

The idea of parallel computer structures is certainly not new. It has already been the subject of numerous research projects and a very vast literature [1]; even a system intended specifically for computational chemistry has been proposed before [2]. By these standards we can claim little in terms of sophistication or originality. In fact, we have deliberately chosen the path of least resistance at many points in the development of this project. This pragmatic approach is in no way critical of more ambitious attempts. Rather, we have different priorities, namely the quick migration of our large-scale scientific applications to parallel execution. This "consumer orientation" is the most distinctive feature of our project.

Many of the characteristics of our parallel strategy follow from these priorities. These characteristics are 1) parallelism based on few (less than 20) but very powerful array processors, with 64-bit hardware; 2) architecture as simple as possible, but extendable; 3) system software that varies as little as possible from that used for normal sequential programming; 4) initial application programming entirely in FORTRAN, since this is the most widely used scientific application language; 5) migration of old sequential code to parallel code with minimal modifications.

It should be noted that all of our current hardware and most of our system software are standard products available "off the shelf"; this is an important factor in the rapid and cost-effective development of our system. Specifically, we have selected the Floating Point Systems FPS-164 for the array processors. This choice is dictated by the fact that these are the only 64-bit array processors currently being marketed. For the host computer, we use an IBM 4341 or IBM 4381. Additional description of our hardware configuration is provided in Section 2.

As already indicated, our principal interest here is not hardware, but rather application software. We want to present our system in *sufficient detail* to facilitate understanding of our experiments in conversion of applications to parallel execution, and to support at least some idea of how other applications might be converted to parallel execution on our system. Thus, in Section 3 we present the strategies we have developed to modify our application programs for effective parallel execution on our system. In order to more precisely understand how these strategies are implemented, we describe the system in Section 4. We include in this section the details of the communication software packages we use, since our system is one of the few places where a FORTRAN-accessible implementation of parallel processing is currently being tested, and we expect our experiments to yield useful insights concerning how such software should look in order to best serve the needs of the user.

Section 5 studies performance of our system in practical applications. Finally, in Section 6, we wrap up our current experience with our system and discuss further developments, particularly those which have to do with our system's serving as a testing ground for parallel execution of application programs.

2. Present configuration

The computer configuration presently working in our laboratory consists of ten FPS-164 attached processors (AP); seven are attached to an IBM 4381 host and the remaining three are attached to an IBM 2914 T-bar connection so that they can be switched between an IBM 4341 host and the IBM 4381 host. The FPS-164 processors are attached to the IBM hosts through IBM three-megabyte-per-second channels available on these hosts. A second IBM 4341, connected to a graphics station, completes the host processor pool. The three IBM systems are interconnected, channel to channel, via an IBM 3880 connector. A schematic diagram of the configuration appears in **Figure 1.**

One attractive feature of this system is the possibility of switching one, two, or three FPS-164s from the IBM 4341 host to the IBM 4381 host. This gives us the flexibility of a "production system" with seven APs and an "experimental" system with three APs. The latter system is used during "prime time" for program development, experimentation with new hardware, debugging of system and/or application programs, etc. During off-prime time and weekends, we typically work with two systems of four APs each and a third system with two APs. Clearly, depending upon user demand, we can use all ten APs for a single job, or, at the opposite extreme, each one of the ten APs independently on ten different jobs.

Each AP contains an independent CPU and its own memory and disk drives. The CPU on the FPS-164 runs at 5.5 million instructions per second, and several concurrent operations can take place on each instruction cycle. In particular, one 64-bit floating-point addition and one 64-bit floating-point multiplication can be initiated each cycle, so that peak performance is about 11 million floating-point operations per second (11 megaflops). Of course, one must make the distinction between peak performance (a characteristic of the machine hardware) and realized

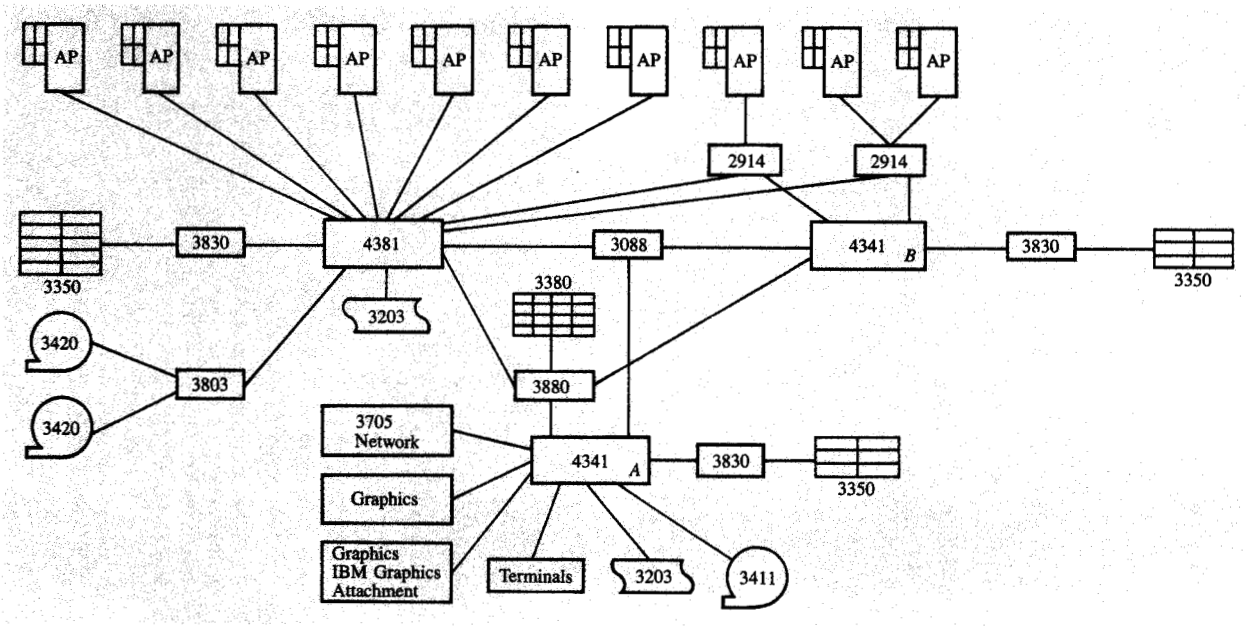

Figure 1

Schematic diagram of the current loosely coupled array of processors. Numbers refer to IBM devices; i.e., IBM 2914 T-bar switch, IBM 4341 and 4381 computers, IBM 3830 and 3880 disk storage control units, IBM 3350 and 3380 disks, IBM 3088 multisystem channel adapter unit, IBM 3203 printer, IBM 3803 tape control unit, IBM 3420 and 3411 tape drives, and IBM 3705 telecommunication network control unit.

performance (depending on the application and the code which implements it as well as the hardware). Nevertheless, the FPS-164 tends to be quite effective for the specialized "number crunching" that constitutes a large part of scientific and engineering computer applications.

Communication between the host computer and the attached FPS-164 is handled by hardware and software provided and supported by Floating Point Systems as a standard feature. An optimizing FORTRAN compiler and supporting utilities (including disk I/O) are also standard products for the FPS-164. The compiler is capable of reasonably effective use of the special architecture of the FPS-164, so that it is practical to run programs written entirely in FORTRAN. An extensive library of subroutines is also provided, and these can be combined with FORTRAN code for additional gains in execution time.

Each of our FPS-164s has at least four megabytes of real random access memory; two have more: eight and ten megabytes, respectively. The memories on the IBM 4341 Model M2 and Model P2 are eight and sixteen megabytes, respectively. The IBM 4381 has sixteen megabytes. Thus, taken as a whole, there is 90 megabytes of real storage available in our system.

Each AP also has four 135-megabyte disks, for a total of 5.4 gigabytes. In addition there are banks of IBM 3350 and IBM 3380 disks accessible to the host computers totaling about 25 gigabytes of disk storage. Tape drives, printers, and communications network interface complete our system.

Very recently, Floating Point Systems announced the FPS-164/MAX. This configuration consists of special-purpose boards that can be added to the FPS-164 to augment performance, particularly on matrix operations. Each MAX board contains two additional adders and two additional multipliers, and so adds 22 megaflops to peak attainable performance. Up to 15 boards can be placed in a single FPS-164, converting it to a machine with a peak performance of 341 megaflops. Special software is required to utilize the MAX boards, and one must carefully examine the extent to which a particular scientific or engineering calculation can be organized to make effective use of this hardware and software.

Acquisition of two FPS-164/MAX boards for each of our ten APs, which is under consideration, would upgrade our system from its current 110-megaflop peak performance to 550 megaflops. Ultimately our system could grow to 3410 megaflops peak capability, but (recalling the distinction between peak performance and realized performance) it is clearly desirable first to explore the gains that one can realistically obtain with only a few 164/MAX boards per AP, so we shall settle at 550 megaflops.

It should be noted that such upgrades have no effect on the parallel programming strategy outlined in the next

section. The strategy is equally effective for APs of any architecture or computational speed. In principle, we could substitute ten vector-oriented supercomputers for our ten FPS-164s, but since a supercomputer costs several million dollars compared with a few hundred thousand for an FPS-164, the cost of this option would be unrealistically high.

In our current configuration, each AP can communicate only with its host computer; there is no possibility of direct communication between two different APs. Furthermore, communication between host and AP is limited by channel speeds. These communication characteristics are important, and motivate us to describe our system as a loosely coupled array of processors (LCAP). We can, of course, accomplish AP-to-AP communication indirectly by having an AP communicate with the host, which then communicates with the second AP. However, communication overhead is a serious consideration in some applications, so the lack of direct AP-to-AP communication is a genuine limitation. Our system would acquire substantial additional flexibility and power if we could move toward a better-coupled configuration in which the APs could communicate with one another directly through a common fast bus and/or shared memory. We will report more on these aspects of our system as soon as data become available.

3. Programming strategy

Several aspects of our approach to parallel programming have already been mentioned in the introduction. Over a period of time our group has developed a substantial collection of scientific application codes, almost entirely written in FORTRAN, and one of the motivations for developing our parallel system was to run these applications for cases that would be too slow and unwieldy on a normal sequential system. Thus we have adopted a simple, pragmatic approach that permits migration of existing sequential code to our parallel system with minimal modification and implements this migration in FORTRAN. Of course, there are some system utilities required (which are accessible through FORTRAN subroutine calls), but we postpone discussion of these to the next section. Here we are concerned with the development of our general programming strategy.

We begin with the observation that large-scale, typically CPU-bound calculations almost invariably involve loops that are traversed many times. Most of the CPU time is consumed in such loops, so that if we adapt the tasks contained in these loops to parallel execution, we find that we actually have most of the code (as measured by execution time) running in parallel.

This is easy enough to accomplish. Let us suppose that our sequential FORTRAN code has a DO-loop of the form

```
DO 500 I=1,N
. . . . . .
. . . . . .  ,
```

with some computational kernel inside the loop (up to statement 500). Then, if we suppose that *NCPU* is the number of APs available for parallel execution, we can keep the same computational kernel and modify the loop to read

```
DO 500 I=ICPU,N,NCPU
. . . . . .
. . . . . .
```

This portion of the program, with the computational kernel and modified loop, is dispatched to each of the *NCPU* APs. Each AP must, of course, have a different value for the index *ICPU*, with $1 \leq ICPU \leq NCPU$.

This rather simple scheme has been applied to all the application programs we have migrated to parallel execution; it was effective in every case. Thus, after migration, typical program flow consists of an initial sequential part handling initial input, setup, etc., followed by a parallel part running simultaneously on several APs. At the end of this portion, the results from the parallel execution must be gathered up and processed by another sequential portion. This may be a prelude to another period of parallel execution, or, ultimately, to development of final results and the end of the run.

There is an obvious limit on this scheme: The computational kernel for a particular value of *I* in the loop example above must not depend on results computed in earlier passes through the loop with a different value of *I*. Our experience so far indicates that this is not a severe restriction; indeed, we find that our code tends to fall naturally into such a form. There are some exceptions, of course, and we have simply left the ones we have encountered in the sequential part of the code. This has had only a mild effect on the overall performance of the resulting program for parallel execution. However, we hope eventually to develop more sophisticated strategies to deal with these cases.

We also point out that this scheme is entirely unaffected by the particular implementation of the computational kernel contained in the loop. One can therefore immediately transfer any improvement in the sequential version of the code to the corresponding parallel version. Nor does migration to a machine with a different architecture affect the scheme, since again only the implementation of the kernel changes.

Programming according to this scheme is presently done "by hand"; there is no software to help with the bookkeeping involved. Eventually we would like to have a compiler and/or an optimizer which could handle some of these tasks and also at least partially automate the use of the communication facilities described in the next section. As a first step, we are considering the possibility of writing a simple precompiler to help with the rudiments of migrating codes from sequential to parallel.

The operation of loading the program and/or data in each AP at the beginning of a parallel portion of the run is overhead that does not appear in the corresponding sequential code; so is the operation of gathering up results from the APs at the end of parallel execution. To safeguard efficiency, these operations should be minimized to the extent possible. The most obvious way to do this is to group the largest feasible portion of code together for a single section to be run in parallel. Thus, in the example above, we dispatch the entire loop to each AP once, rather than dispatching the contents of the loop each time we pass through it in the code. Moreover, the loop to which this treatment is applied should be the outermost one, to the extent possible. Sometimes further economies can be gained by rearranging the code to yield fewer but larger single tasks to dispatch to the APs.

One should also carefully control the amount of data to be transmitted to the APs and the amount of data to be gathered from the APs at the end of a parallel part of the program. When this is done, one frequently finds large intermediate arrays that are used by the parallel part of the program but need never be referenced in the sequential part. Sometimes it is necessary to recreate such arrays each time the AP runs one of the parallel tasks, but additional economies are available if the array can be maintained on the AP during the entire program run, spanning all the parallel tasks. In our molecular electronic structure code [3], this occurs for the two-electron integrals, which are parceled out among the disks attached to the APs and read in as required in parallel. For our Monte Carlo code [4] and our molecular dynamics code [5], tabular material is set up in AP memory at the beginning of the run and remains without modification, to be used each time molecular interactions are calculated in parallel.

One can break into two parts the total execution time T for an application program running in parallel, so that $T = T_s + T_p$, where T_s is the total execution time for sequential parts of the code and T_p is the time spent on parallel execution. Since there are usually several sequential portions in a run, we write $T_s = \Sigma_i t_{is}$ where the index i numbers the different sequential portions. Similarly, we give $T_p = \Sigma_j t_{jp}$, where t_p is the time between dispatch of the parallel task to various APs and the subsequent gather. To be more specific, say that an AP indexed by α takes time $t_{j\alpha}$ for its part of the parallel execution. We then have a collection of times $t_{j\alpha}$, one for each AP, and t_{jp} is the longest of these. Ideally, T_s should be much smaller than T_p, because no more than one AP is used during this time; the others are idle. For the same reason, each of the $t_{j\alpha}$ should be identical to all the others, or nearly so; any difference means that at least one AP is standing idle part of the time.

In practice, the requirement that all $t_{j\alpha}$ be identical cannot be achieved exactly. In our loop example, we need N exactly divisible by *NCPU*, and this is not a normal occurrence in general application runs where N (and possibly also *NCPU*) varies from case to case. Moreover, having N exactly divisible by *NCPU* does not guarantee an ideal case, because the computational kernel may vary in run time for different values of I (an example, which actually occurs in our SCF code, would be stepping through an array where the calculation is skipped when the array element is zero). Sometimes it is possible to rearrange the computation in terms of a new loop variable [4] to get the $t_{j\alpha}$ more nearly identical, but this strategy has its limits, since it usually involves developing more elaborate loop control code, and executing this code takes time.

In our loop example, the difference among the various $t_{j\alpha}$ is essentially the execution time for one execution of the computational kernel. This is a measure of the granularity of the problem. We see at once that the ratio of the smallest $t_{j\alpha}$ to t_{jp} tends to approach 1 as N increases. Moreover, the ratio t_{jp}/T_s is expected to increase as N increases. This exemplifies a general tendency: As calculations get larger, the parallel program becomes more efficient. Of course this is just the trend we would like to see, since parallelism is intended to handle cases too large for effective sequential execution.

We note that we have arrived at a rough criterion by which we can judge whether a particular task can be effectively migrated to parallel mode: t_{jp} must be substantially greater than the sum of the lag time associated with granularity and the transmission time for the task. An interesting effect emerges when we consider how the situation changes as more APs are added. This decreases t_{jp} and increases transmission time, while the granularity remains constant. One concludes that, for a given application, parallel execution becomes less efficient as the number of APs increases. This effect is actually observed, so that the ability to vary the number of APs used in a particular application run is important. Our system has this flexibility, and all our parallel programs incorporate it.

4. System considerations

As already indicated in the introduction, we are not concerned with development of elaborate system software to make our parallel system work. Instead, we use, to the extent possible, "off-the-shelf" software. This approach has the merit of reducing our programming overhead very substantially and permitting us to use proven software immediately in our applications.

On the IBM hosts, our operating system is the IBM Virtual Machine/System Product (VM/SP). For the APs, we use the software provided by Floating Point Systems for hosts running under this system. We have not found it necessary to modify either set of software in order to run our applications in parallel.

VM/SP is a time-sharing system in which jobs run on virtual machines (VM) created by the system; these VMs simulate real computing systems. The standard software

provided by Floating Point Systems for use on the FPS-164s embodies the restriction that only one AP can be attached to a VM. Of course, for a task running in parallel, more than one AP is required. Our solution to this is to introduce extra "slave" VMs to handle the extra APs we need. To make this work, one must have a way to communicate between different VMs; this is provided by the Virtual Machine Communication Facility (VMCF), which is a standard feature of VM/SP [6].

In general, then, a parallel task consists of several FORTRAN programs, each running on a separate VM in the host system, and each controlling a particular AP on which additional FORTRAN code runs. On one of the VMs, the "master," is the part of the original FORTRAN code intended to be run on the host, combined with utility subroutines that handle communication with the slave VMs and with the AP attached to the master VM. The programs running on the slave VMs are much shorter, since the slave VMs are nothing more than transfer points for communication between the master program and the APs attached to the slaves. In essence, the slave VM programs consist of code to handle communication with the master and utility subroutines to communicate with the AP attached to the slave.

Since each VM is attached only to a single AP, the standard utilities provided by FPS [7] for communication between host and AP can be used without modification. We have already described in the previous section the development of code to run in parallel on the APs. As we saw there, the code running on the various APs (including the one attached to the master) is identical for each AP. Accordingly, the programs running on each of the slave VMs are also identical; it is up to the master program to see that the slaves each get different data so that they control distinct calculations.

It remains to describe the utilities that handle communication between master and slave VMs. These utilities are not "off-the-shelf" in the same way as the utilities for communication between host and AP. As already mentioned, the vehicle for communication between VMs is provided as part of the VM/SP system, namely VMCF, so that no real system programming is necessary. However, use of VMCF requires calls to the system from assembler code. It is desirable to package this code, once and for all, in utility subroutines that can be invoked from normal FORTRAN code; this considerably simplifies program migration to parallel mode. Development of this set of utilities, which we call VMFACS (Virtual Machine FORTRAN-Accessible Communications Subroutines), was one of the first steps in implementing our parallel system, and it has required virtually no subsequent modification. It is therefore of interest to describe VMFACS in detail, since it is one of the very few packages to make parallel processing accessible to the FORTRAN programmer, and our experiences with it can be expected to be helpful in discovering how such packages can best meet the needs of the FORTRAN programmer.

One of these VMFACS utilities packages, SLVTEL, is used by the slave VMs. This package contains four different FORTRAN-callable subroutines. The first of these is invoked by a call of the form

```
CALL SLVTEL (MASTER,RCVADD,RCVLEN,
ERRNUM,ERRBRN)
```

Here MASTER is an eight-byte character constant giving the userid of the slave VM's master, RCVADD is the beginning address of the data the slave is expecting to receive from the master, and RCVLEN is a four-byte integer giving the length (in bytes) of the data. ERRBRN indicates the FORTRAN statement which is the error return for the subroutine, and ERRNUM is a four-byte integer where a description of any error is placed by the SLVTEL package. This call connects the slave VM to VMCF so that the master can communicate with it, and then causes the VM to wait until the master sends the data the slave is expecting to receive in RCVADD. When the data have been successfully placed in RCVADD, control returns to the calling program. One effect of this call is to set ERRNUM aside for error recording by the SLVTEL package, so the user should be careful not to alter ERRNUM while the package is in use. Of the possible error conditions, the most likely is an attempt by the master to send data having a length different from the length indicated by RCVLEN.

After the slave VM receives its initial data from the master VM, it performs some task using the data and eventually reports the result to the master. This is accomplished by a call of the form

```
CALL SNDRCV (SNDADD,SNDLEN,RCVADD,
RCVLEN,ERRBRN) ,
```

where SNDADD is the beginning address of the data to be sent to the master and SNDLEN is the integer length of the data (in bytes). As in the call to SLVTEL, RCVADD is the beginning address of the data expected from the master, RCVLEN is the length of the data, and ERRBRN points to the error return. The address of ERRNUM is retained from the SLVTEL call, and this integer again contains a description of any error encountered. After the data in SNDADD have been sent to the master, SNDRCV waits for data to be sent from the master to the slave for the slave's next task. When the data have been successfully placed in RCVADD, control returns to the calling program.

The last task performed by the slave is simply the orderly exit from the slave program. This does not involve sending anything to the master but should include disengagement from VMCF. The call to perform this disengagement is of the form

```
CALL SLVOUT (ERRBRN) ,
```

where ERRBRN again points to the error return. It should be noted that an error return occurs not only for errors encountered in executing SLVOUT, but also for errors recorded for previous operations using the SLVTEL package.

VMCF is also used by the utilities provided by FPS to attach an AP to or detach it from a VM, and this alters the VMCF setup used by the SLVTEL package. To deal with this contingency, the SLVTEL package contains one more call, of the form

CALL RSLVTL (ERRBRN) ;

here ERRBRN once again points to the error return. This call merely restores the VMCF setup which the call to SLVTEL initiated.

Use of the SLVTEL package requires information about only one other VM, namely the master. On the other hand, the VMFACS package for the master VM, MASTEL, must maintain information on all the slave VMs attached to the job; hence it is necessarily somewhat more complex. This package is initiated by a call of the form

CALL MASTEL (N,SLVUID,ADDR,IND,ERRNUM, ERRBRN)

Here N is a four-byte integer giving the number of slaves attached to the run, and SLVUID is an array of eight-byte character constants set up so that SLVUID(I) is the userid of the Ith slave VM. ADDR and IND are, respectively, eight-byte and four-byte work arrays set aside for the use of the MASTEL package; both arrays must have at least N elements. As in the SLVTEL call, ERRNUM is the integer describing any error condition, and ERRBRN gives the return point when an error occurs. It is important to point out that the arrays SLVUID, ADDR, and IND are used to help the MASTEL package keep track of its transactions, so these arrays must not be altered while the package is active. The same comment applies to ERRNUM, since a description of errors is kept there by the entire MASTEL package.

The call used to send data from the master to one of the slaves has the form

CALL SNDRCV (I,SNDADD,SNDLEN,RCVADD, RCVLEN,ERRBRN) ,

where I indicates the Ith slave VM and the remaining arguments are used the same as in the SNDRCV call for the SLVTEL package. However, the SNDRCV call in the MASTEL package causes a behavior considerably different from that caused by the SNDRCV call in the SLVTEL package. In the MASTEL package, CALL SNDRCV does not cause a wait to occur until data arrive in the array beginning with RECADD; in fact, there is not even a wait for the send transaction to be completed, so one can begin sending to another slave immediately. Consequently, one must check to see that the send transaction has actually been completed before disturbing the SNDADD array.

To check the status of transactions with the slaves, there is a call of the form

CALL MWAITO (INDFLG,ISLV,ERRBRN) ,

where INDFLG is a four-byte integer array controlling the inquiry to be made, ISLV is the four-byte integer returned by the call, and ERRBRN gives the error return point. INDFLG should contain at least as many elements as there are slaves. The integers in the INDFLG array must correspond to the following requests:

−1 skip testing for that slave;
0 test for completed transaction; that is, slave ready for next SNDRCV (and the response from any previous SNDRCV is available in the appropriate RCVADD array);
1 test for send to slave which is not complete;
2 test for completed send transaction to slave, but with no reply from slave available in RCVADD array.

MWAITO tests the status of the slaves as requested and returns the index of the first slave it finds with the requested status as the value ISLV. If no slave has the status requested, a wait occurs until a slave with the requested status can be found, whereupon the index of that slave is again returned in ISLV. Errors causing an error return originate from some SNDRCV transaction rather than from the possible wait. Some care must be exercised in setting up the INDFLG array for a MWAITO call, since an indefinite wait occurs if none of the specified conditions ever becomes true. It is safe to have at least one element of INDFLG set to 0, and this normally constitutes the most useful test. The test values 1 and 2 are not so safe, since the MASTEL package proceeds with the transaction while the tests are in progress, and this process may happen too fast for useful tests. If one can be confident that the task being performed by the slave will take enough time so that it will not suddenly finish and report back, the test value 2 can be useful to wait for the SNDADD array to become available for other use. The test value 1 is included for completeness but is not normally useful, because the send part of the transaction occurs very quickly.

The status of transactions with the slaves can also be checked by another call, which has the form

CALL CLRTEL (ERRBRN) ,

where, as usual, ERRBRN indicates the error return. This call waits for pending send or receive transactions to be completed, which corresponds to testing every slave to see that it does not have test value 1 (either test value 0 or test value 2 are acceptable). This call is the most convenient way to ensure that the last data to be sent to the slaves have actually been sent before the master disengages from VMCF prior to the end of the run for the master program. One should always do this, because premature disengagement

from VMCF by the master prevents the slaves from receiving their data, causing an error to emerge from the SLVTEL package.

Finally, the master disengages from VMCF and terminates the use of the MASTEL package by a call of the form

```
CALL  MASOUT (ERRBRN)  ,
```

where ERRBRN again points to the error return.

It should be noted that the MASTEL package does not actually start the slave programs running. This is done (on the appropriate VM) by the person submitting the run at the same time the master program is started. In reality, all the slave programs should be started slightly before the master program so that they are ready when the master program attempts to send to them.

The VMFACS utilities were designed to provide the required functions in the simplest fashion that would provide sufficient flexibility. The result, especially for the MASTEL package, is to provide only those functions that cannot readily be provided by FORTRAN coding. For example, several variants of the test/wait scheme provided by MWAITO might be useful, but the FORTRAN programmer is obliged to develop them on his own. Also, there is no attempt to facilitate recovery from possible errors in using these packages. This is deliberate, in order to allow, and even encourage, the FORTRAN programmer to try different strategies in migrating code to parallel execution.

The VMFACS utilities packages are experimental, as is our entire parallel system. They may be expected to change as our system undergoes further development. One possibility is to replace or supplement the use of VMCF with the Inter-User Communications Vehicle (IUCV), another feature of VM/SP [6]. Another possibility under active consideration is an MVS alternative to our VM/SP system. Ultimately, one might expect to eliminate the need for these communication packages by replacing the standard FPS utilities with utilities that would allow many APs to be attached and controlled by a single VM.

On the other hand, these communications packages serve us rather well. Our tests indicate that VMCF communications are no great burden on the system: They amount to a small fraction (about one tenth) of the time required for channel transmission between host and AP. Clearly, substantial improvement in communication requires improvement in our hardware, such as the common fast bus or shared memory mentioned in Section 2.

An interesting sidelight of these developments emerges when the code running on the APs is incorporated into the slave or master programs, as appropriate, to yield an application running entirely on the host but spread across several VMs. This strategy is sometimes useful in developing and debugging code to be run in parallel. For production runs on a single-CPU machine this approach is pointless, since it simply introduces additional system overhead, but this is no longer the case for runs on a multiple-CPU machine such as the IBM 3081. In this case, different VMs can run on different CPUs, so the run becomes a parallel one. Our tests of some of our parallel applications on an IBM 3081 under VM/SP demonstrate that the expected parallel performance is actually achieved. Our parallel applications have also been run under MVS [8] on an IBM 3084, and again the expected parallel performance was fully achieved. In this case, the subroutines described above were adapted to use MVS system facilities instead of VMCF. Since the adaptation leaves the communication strategy unaltered, the MVS implementation can execute the same FORTRAN code used for the trials under VM/SP.

5. Practical tests

Our parallel system has developed with large-scale scientific applications programs waiting to migrate to the system as soon as possible. In fact, an important motive in development of the system was to use it to extend the range of our research in theoretical chemistry and biophysics. Applications embodying a fairly broad range of computational demands are now running on the system, and this provides a basis for very realistic tests of its performance.

We note that the first test of our system was in fact the attempt to migrate existing applications to the system. As already indicated, our system developed with the idea that migration of old sequential code to parallel code should not require extensive code modification, and we have in fact followed this path. It is important to note that much of our applications code was developed within our laboratory, so we know the code well enough to implement effectively the strategies outlined in Section 3. Provided this requirement is satisfied, we have found that migration of good functional sequential code to our parallel system is not much more difficult than migration of such code from one sequential computer system to another sequential computer system with a different architecture. We can conclude that our system is quite accessible, with good "user-friendliness."

We consider here four applications currently running on our system. Two of them, namely the integrals program and the SCF program, are parts of our quantum molecular program package. In addition, we have Monte Carlo codes for simulation of statistical mechanics of liquids and solutions, and molecular dynamics codes which provide time-dependent simulations of liquids and solutions. Each of these applications places different demands on our system, and it is useful to describe these demands in more detail.

The *integrals program* repeatedly evaluates algebraic expressions with various sets of parameters. Evaluation of any one integral is fast, but any complete run includes huge numbers of these integrals. The computed integrals are stored on disk files which can be larger than a gigabyte in size. Computation of any integral is independent of any

other integral, which makes it easy to run the integrals program in parallel; one simply decides how the set of integrals is to be divided among the available APs. Once the run is set up, there is no communication between the tasks running on the different APs, and even disk I/O takes place entirely on the AP (parallel I/O). The time a particular task runs without interruption on the AP is the elapsed time for the entire run (a unique feature of this application), and this is typically on the order of hours. The only obstacle to ideal parallelism is that we have not found a way to divide the set of integrals in the run that can guarantee perfectly even distribution among the APs.

The *SCF program* iteratively improves on an initial guess for molecular electronic structure until convergence is achieved. Each iteration can be divided into several steps, of which by far the most time-consuming is combining results from the last iteration with the integrals file generated by the integrals program to develop the current iteration. This step is the only one running in parallel at present. It involves heavy disk I/O (which again takes place in parallel) and also very substantial CPU time. The time for this step running in parallel is typically a fraction of an hour and requires transmission of several hundred kilobytes of data between host and AP. A more detailed description of both the integrals program and the SCF program can be found in a previous report [3].

Our *Metropolis-Monte Carlo programs* deal with liquids or solutions one molecule at a time, and the main task is evaluation of the change in the potential energy of the bulk each time a molecule is moved. It is this task that runs in parallel. All other tasks are so fast that they cannot benefit from parallel execution. There are several versions of our Monte Carlo code, depending on how elaborate (and hence realistic) an energy expression is being used. Thus, the time a task runs in an AP without interruption varies from a fraction of a second up to several seconds. Data transmission between host and AP is on the order of a hundred bytes per task. For more detail concerning our Monte Carlo programs, the reader is referred to our previous report [4].

Our *molecular dynamics* programs simulate the kinetic motion of molecules in bulk liquid or solution over a period of time divided into many time steps, with each time step involving the evaluation of many molecular energies and forces. This is the bulk of the computation, although we also compute the resultant molecular motion in parallel for each time step. Again, there are several versions of our molecular dynamics code, depending on the energy expression being used. The typical time for a task run in an AP without interruption is a fraction of a minute, and this involves data transmission between host and AP of as much as a few megabytes. We again refer the reader elsewhere [5] for additional details of these programs.

We have timed specific application runs for all four of these programs running sequentially (one AP), and running in parallel on three APs, six APs, and ten APs. One objective, of course, is to see how successful we are in exploiting our parallel system. We would also like to see how close we come to supercomputer performance, and so we have run these applications on a CRAY 1S as well. The application code for the CRAY was developed under constraints analogous to the constraints for our parallel application codes, that is, the minimum modifications required to run properly under that system. Efforts to modify the code to better exploit the vector architecture of the CRAY would certainly have resulted in faster timings for that machine, but, conversely, we could gain on our parallel system by adapting our code to the architecture of the FPS-164. As they stand, we believe our results are useful, even though they cannot be regarded as anything like a definitive comparison between the two systems.

Our timings can be found in **Table 1.** We find that for the integral, Monte Carlo, and molecular dynamics programs, the execution time on our system with six APs almost equals that for the CRAY 1S. For the SCF program, results are not as good. We can attribute this to the sequential part of the SCF code, which grows in significance as more APs are used for the parallel part. We have already started improvement of the SCF code, and we expect that this will bring its performance into line with the other applications. In all cases, we see some degradation from full parallelism. For example, for the integrals run with 42 atoms we would expect the three-AP result to be one third the execution time with one AP, 67.8 minutes. The actual execution time for three APs is 68.9 minutes, so we have a little over a minute of "overhead time" in this case. Additional overhead shows up as one progresses from three to six APs or from six to ten APs. The causes of this overhead, and the strategies that can be used to minimize it, have already been discussed in Section 3. In addition, we are investigating hardware modifications which should reduce system and communications overhead.

6. Discussion

Although our experimental parallel system has been in existence for only a rather short time, we believe it has already established itself as a pragmatic answer to many computer-intensive problems. The scientific applications that have so far migrated to our system have already opened up new vistas in our research in theoretical chemistry and biophysics [3–5, 9, 10]. There is no reason not to expect similar benefits in other research areas. On the contrary, the variety of our applications delineates the flexibility and versatility of our system, and parallelism appears to be a technique of broad applicability in the physical sciences and related engineering fields.

In order to better appreciate this, let us examine why the parallel programming strategy presented above, particularly in Section 3, should be as effective as it proves to be. Of

Table 1 Comparison of execution times for our system with different numbers of APs versus the time needed on the CRAY 1S (applications code not optimized for either system), in minutes. Measurements for up to six APs were performed with an IBM 4341 host, except for the molecular dynamics runs; all runs with ten APs took place with an IBM 4381 host.

Job	*One AP*	*Three APs*	*Six APs*	*CRAY-1S*	*Ten APs*	*Host*
Integrals (27 atoms)	71.7	24.0	12.3	10.6	7.8	4341
SCF (27 atoms)	46.7	21.0	17.5	8.6	12.0	4341
Integrals (42 atoms)	203.4	68.9	38.3	32.3	21.2	4341
SCF (42 atoms)	108.5	44.9	34.1	19.6	22.0	4341
Integrals (87 atoms)	2163.0	730.0	380.0	309.0	247.0	4341
Monte Carlo	162.1	57.8	32.0	28.4	22.0	4341
Molecular dynamics	87.1	38.3	23.3	20.1	—	4381

course, the simplest and perhaps the most natural way an application run can achieve large scale and long run times is by repeatedly passing through a loop with an extensive computational kernel, but this is not enough to ensure that our strategy will be successful. It is also necessary that the kernels on different passes through the loop be sufficiently independent of one another to be amenable to parallel computation in a simple way. In our molecular dynamics and Monte Carlo codes, parallel computation centers on interactions among the particles in a system consisting of n particles. In general, the computation of a given interaction term (such as a two-body interaction between a particular pair of particles) is independent of the computation of any other distinct interaction term (e.g., another two-body interaction between a different pair of particles). This yields the computationally independent kernels we use to implement parallel execution. The same situation occurs in our calculations of molecular electronic structure, except that electrons replace molecules as the particles in question, and quantum mechanics applies instead of classical mechanics. We may conclude that the type of computational independence we have exploited in our applications is also characteristic of the closely related computational simulations in solid state physics, and, more generally, in any area of the physical sciences where simulations track the motions of some group of discrete particles.

We can bring no such experience to the area of physical simulations of continuous media. However, we note that there is much activity in this area by other investigators, and recent work indicates good prospects for successful application of parallel techniques. In addition, we look for parallel computations to be applied in other fields, including econometrics, graphics, etc., where computational techniques analogous to those in the physical sciences are employed.

In order to study such questions in more depth, and also to learn more about how to use our system, we wish to extend our work to encompass applications from many other fields. For this reason, we are implementing a "visiting scientists" program where scientists from universities and other research institutions will be able to investigate at our laboratory the possibility of adapting parallelism to other scientific problems. In a few years, we expect these studies to accumulate a much more comprehensive understanding of the practical implementation of parallelism than is possible at present.

We conclude by stressing the rather exceptional versatility of our system (scalar, vector, and parallel), its flexibility, its high performance, and its "user-friendliness." We note also its high reliability: Any one of the IBM hosts can be used to enter our system and it is most unlikely that all ten APs will be down at the same time. Finally, the realistic possibility of increasing peak performance from the current 110 megaflops to 550 megaflops and eventually even much higher, with a corresponding increase in practical performance, opens the door to computational research that was previously unreachable. However, this optimism should be tempered by the realization that we are only at the beginning of our explorations, and many aspects, both in hardware and in software, may prove to be more of an obstacle than we expect.

Acknowledgment

It is our pleasure to thank E. Clementi for his helpful comments on this paper and for his directions; and D. Meck and H. Khanmohammadbaigi for many discussions on parallel programming. We also wish to thank Arthur G. Anderson and Earl F. Wheeler for their interest in our effort aimed at hardware and software parallelism.

References and note

1. (a) *Proceedings, International Conference on Parallel Processing*, R. H. Kuhn and D. A. Padue, Eds., Bellaire, MI, August 25–28, 1981. (b) R. W. Hockney, in *Parallel Computers: Architecture, Programming and Algorithms*, R. W. Hockney and C. R. Jesshope, Eds., Adam Hilger, Ltd., Bristol, UK, 1981. (c) *Parallel Processing Systems*, J. Evans, Ed., Cambridge University Press, New York, 1982. (d) Y. Wallach, "Alternating Sequential/Parallel Processing," *Lecture Notes in Computer Science* **124**, Springer-Verlag New York, 1982.
2. K. R. Wilson, in *Computer Networking and Chemistry*, P. Lykos, Ed., *ACS Symposium Series* **19**, American Chemical Society, 1975.

3. E. Clementi, G. Corongiu, J. Detrich, S. Chin, and L. Domingo, "Parallelism in Quantum Chemistry: Hydrogen Bond Study in DNA Base Pairs as an Example," *Int. J. Quant. Chem.* (Quantum Chemistry Symposium) **18,** 601 (1984).
4. J. H. Detrich, G. Corongiu, and E. Clementi, "Monte Carlo Liquid Water Simulations with Four-Body Interactions Included," *Int. J. Quant. Chem.* (Quantum Chemistry Symposium) **18,** 701 (1984).
5. H. L. Nguyen, H. Khanmohammadbaigi, and E. Clementi, *J. Comput. Chem.* (in press).
6. *Virtual Machine/System Product System Programmer's Guide, Third Edition*, Order No. SC19-6203-2, available through IBM branch offices.
7. *FPS-164 Operating System Manual, Vols. 1–3*, Publication No. 860-7491-000B, Floating Point Systems, Inc., Beaverton, OR, January 1983.
8. D. L. Meck, "Parallelism in Executing FORTRAN Programs on the 308X: System Considerations and Application Examples," *Technical Report POK-38*, IBM Information Systems and Technology Group, Poughkeepsie, NY, April 2, 1984. For another set of FORTRAN-callable communications subroutines to support parallel execution on the IBM 308X under MVS, see IBM Program Offering 5798-DNL, developed by P. R. Martin; the Program Description Operations Manual for this program offering is Order No. SB21-3124 (release date May 4, 1984).
9. J. H. Detrich, G. Corongiu, and E. Clementi, "Monte Carlo Liquid Water Simulation with Four-Body Interactions Included," *Chem. Phys. Lett.* **112,** 426 (1984).
10. E. Clementi, G. Corongiu, J. H. Detrich, H. Khanmohammadbaigi, S. Chin, L. Domingo, A. Laaksonen, and H. L. Nguyen, "Parallelism in Computational Chemistry: Applications in Quantum and Statistical Mechanics," in *Structure and Motion: Membranes, Nucleic Acids and Proteins*, E. Clementi, G. Corongiu, M. H. Sarma, and R. H. Sarma, Eds., Adenine Press, Guilderland, NY, 1984.

Received September 7, 1984; revised October 31, 1984

Giorgina Corongiu *IBM Data Systems Division, Neighborhood Road, Kingston, New York 12401.* Dr. Corongiu received the doctoral degree in theoretical chemistry from the University of Pisa, Italy, in 1976. Her doctoral dissertation was on quantum mechanical interaction potentials for amino acids and water molecules. In 1977 she joined the G. Donegani Research Institute in Novara, Italy, and later came to the United States, working from 1979 to 1982 at IBM Poughkeepsie with a fellowship from the National Foundation for Cancer Research. In 1982, she joined IBM at Poughkeepsie, moving in 1984 to Kingston. Dr. Corongiu's research interest is in computer simulation of complex chemical systems, in particular biological systems, using both quantum and statistical mechanics. Her scientific work is documented in more than 40 publications in international journals. Since 1983 she has contributed to the development of the parallel research project for scientific engineering applications. Dr. Corongiu is a member of the American Chemical Society, the American Physical Society, the International Society of Quantum Biology, and the New York Academy of Science.

John H. Detrich *IBM Data Systems Division, Neighborhood Road, Kingston, New York 12401.* Dr. Detrich received his Ph.D. degree in theoretical physics from the University of Chicago in 1971, and has held postdoctoral fellowships at the University of Chicago and the University of Wisconsin. He held a National Academy of Sciences–National Research Council Senior Resident Research Associateship at the NASA Langley Laboratory in Hampton, Virginia, from 1981 to 1983. Dr. Detrich joined IBM in 1983. His current interests include development of the LCAP parallel processing configuration and computer simulations of biological systems using quantum mechanics and statistical mechanics.